J. Mod. Power Syst. Clean Energy (2013) 1(2):159–166
DOI 10.1007/s40565-013-0024-0

# Large-scale branch contingency analysis through master/slave parallel computing

Xi YANG, Cong LIU, Jianhui WANG (✉)

**Abstract** Contingency analysis (CA) requires fast execution time for real-time power system operations. Because CA problems can naturally be divided into separate subtasks, parallel computing helps to speed up the computation time. This paper proposes a master/slave parallel computing architecture and studies the computation of CA in a large-scale power system through high performance computing, adopting a message passing interface for implementation. In particular, although the execution time of CA varies, there is a tradeoff between having an imbalanced workload and "paying" a synchronization penalty for parallel computing: either factor blocks the progress of scalability. The proposed layered dynamic scheduling method is effective to tackle the challenge of high synchronization cost and workload imbalance and have the potential to further scale for the $N-2$ contingency analysis.

**Keywords** Massive parallel computing, Power system, Contingency analysis

## 1 Introduction

Contingency analysis (CA) is performed to determine whether steady-state operating limits would be violated when credible contingencies occur. CA consists of two parts: contingency selection and contingency evaluation [1–4]. In theory, an extraordinarily large number of possible contingencies can occur at any moment in a large-scale power system, and the window of time, during which system operators can analyze the event and take appropriate preventive (pre-contingency) and corrective (post-contingency) actions [5, 6], is quite limited. If the contingency selection is too conservative, its analysis takes too long time; if the selection is not conservative enough, critical contingencies causing constraint violations or catastrophes could be missed. Thus, a solid contingency selection procedure followed by contingency evaluation for a set of selected contingencies must be executed for proper control actions.

Today, having the ability to perform rapid CA for large-scale systems is critical for safe and reliable power grid operations. In general, such a reliability evaluation should be carried out within a few minutes. Moreover, for a single CA task, the computational time increases much faster than that of the linear growth with the scale of the power system, due to the non-linearity of power flow computation.

As shown in Fig. 1, the CA problem could be decomposed as independent subtasks and solved in parallel, which significantly shortens the computation time. If the execution time for individual tasks is approximately the same, the workload is in balance. We could assign tasks to processors deterministically, thus avoiding synchronization overheads. Otherwise, if the power flow computation in different contingency cases may take different time in most cases, in order to achieve workload balance, we would need to schedule the tasks to the processors dynamically and synchronization costs could hardly be avoided. In addition, the design of the CA algorithm is also important in accelerating the entire process.

Reference [7] demonstrated parallel computing methods to accelerate CA and [8] computed CA for the $N-X$ CA by high-performance computing. Furthermore, [8] and [9]

Received: 3 September 2012 / Accepted: 16 April 2013 / Published online: 7 September 2013

X. YANG, C. LIU, J. WANG, Decision and Information Sciences Division of Argonne National Laboratory, Argonne, IL 60439, USA
(✉) e-mail: jianhui.wang@anl.gov
X. YANG
e-mail: xyang34@hawk.iit.edu
C. LIU
e-mail: liuc@anl.gov

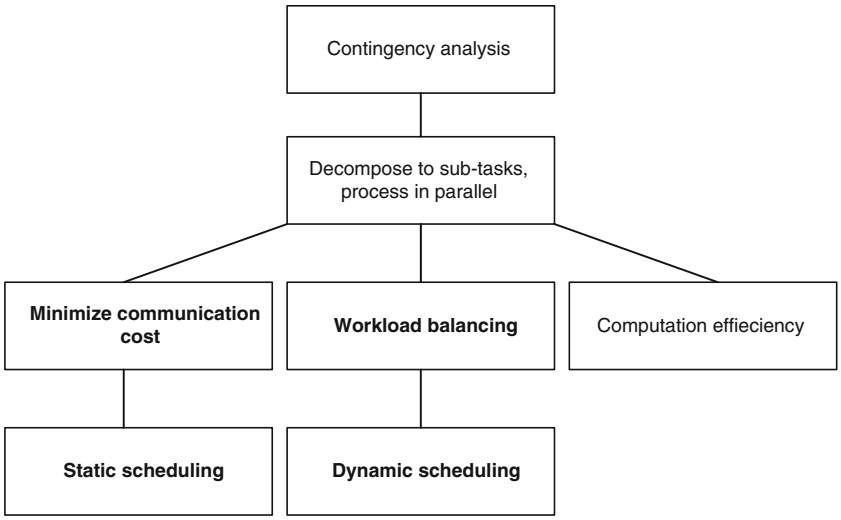


**Fig. 1** Converting CA to parallel computing problems

applied a single, counter-based scheduler that would implement CA task scheduling and dynamically assign tasks according to the current availability of computational processors. They demonstrated an excellent speedup at the 512 process scale. The work also revealed that single-point, counter-based dynamic task scheduling could be a bottleneck, and consequently employed a multicounter-based scheduling method, in which tasks are re-allocated from a heavy to a light workload group. This "master/slave" approach is a typical paradigm for parallel computing and an ideal model for addressing the single-program multiple-data (SPMD) problem [10, 11]. Reference [12] discussed static and dynamic parallel scheduling for execution times of various tasks underlying a heterogeneous platform. Recently, a "superior/master/slave" hierarchical topology for task assignment is presented in [13]. Reference [13] revisited the classic master/slave solution to schedule embarrassingly parallel tasks, which is the master for scheduling tasks heuristically. Various task-process mapping algorithms are studied according to input data patterns. By simulation, some guidelines are given to choose a data distribution topology and a mapping algorithm.

This paper adopts parallel computing techniques to help speed up large CA. In the master/slave architecture, the I/O workload is minimal and is handled with the master reading the data, and then broadcasting to the slaves. It focuses on the challenges of workload scheduling, workload balance, and limiting the communication cost of CA computation. Our contribution is to address this large-scale massive parallel computing problem of CA, apply the hierarchical task scheduling, which efficiently limits the synchronization penalty along with the dynamic task scheduling strategy, and introduce the method to $N-2$ large-scale contingency analysis with robust scalability.

The rest of the paper is organized as follows. CA algorithms are introduced in Sect. 2. Section 3 presents parallelism for CA and parallel implementation. It also reveals the limitation of asynchronous nature of large-scale CA, and correspondingly derives a group-based scheduling method to address the scalability problem. Then, considering the scenarios of various workloads for CA tasks, dynamic scheduling is used to solve the problem and demonstrate its limitation. After profiling and analyzing the bottleneck of counter-based dynamic scheduling, a three-layered scheduling approach is proposed that succeeds the trade-off of synchronization cost and workload balancing, which processes massive CA problems on a large scale. Section 5 presents conclusions of this study.

## 2 Algorithms

The (1)–(4) show the basic $PQ$ fast decoupled power flow calculations [14]:

$$\Delta\theta_k = [B']^{-1}\frac{\Delta P(\theta_k, V_k)}{V_k} \tag{1}$$

$$\theta_{k+1} = \theta_k + \Delta\theta_k \tag{2}$$

$$\Delta V_k = [B'']^{-1}\frac{\Delta Q(\theta_{k+1}, V_k)}{V_k} \tag{3}$$

$$V_{k+1} = V_k + \Delta V_k \tag{4}$$

where $B'$ and $B''$ are fixed Jacobian matrices determined by the admittance parameters of a power network; $\theta$ is the voltage phase angle of buses; $V$ is the voltage amplitude; $\Delta Q$ and $\Delta P$ result from nodal power flow equations. When using sparse-matrix oriented solution methods to solve power flow equations, a large amount of CPU time will be spent on the triangular decomposition of matrices $B'$ and

$B''$. Therefore, in order to avoid forming ordered triangular factorization repeatedly after branch outages, we utilize compensation techniques to obtain the solution of linear equations. Actually, branch outages can be considered in an efficient way by adding an additional part, $\Delta B'$and $\Delta B''$, into the previous $B'$ and $B''$as follows:

$$\begin{aligned}\Delta\theta_k &= [B' + \Delta B']^{-1} \frac{\Delta P(\theta_k, V_k)}{V_k} \\ &= \left[B' + M\delta y' M^{\mathrm{T}}\right]^{-1} \frac{\Delta P(\theta_k, V_k)}{V_k}\end{aligned} \tag{5}$$

where $\delta y'$ is the $m \times m$ matrix, it contains correction information for $B'$; $m$ is number of branch outages; $M$ is the $n \times m$ incidence matrix, which relates to outages. Generally, for one branch outage, we have

$$M^{\mathrm{T}} = [0, \ldots, 1, \ldots, -1, \ldots, 0] \quad M_p = 1,\ M_q = -1$$

Line $p$–$q$ without transformer, or

$$M^{\mathrm{T}} = [0, \ldots, 1, \ldots, -N, \ldots, 0] \quad M_p = 1,\ M_q = -N$$

Line $p$–$q$ with transformer, Tap $= N$ ($p$ side).

According to the matrix modification inverse lemma (MMIL), (5) can be changed as:

$$\Delta\theta_k = \left([B']^{-1} + [B']^{-1} M C' M^{\mathrm{T}} [B']^{-1}\right) \frac{\Delta P(\theta_k, V_k)}{V_k} \tag{6}$$

where

$$C' = \left([\delta y']^{-1} + M[B']^{-1} M^{\mathrm{T}}\right)^{-1} \tag{7}$$

so

$$\begin{aligned}\Delta\theta_k = {} & [B']^{-1} \frac{\Delta P(\theta_k, V_k)}{V_k} + [B']^{-1} M([\delta y']^{-1} \\ & + M[B']^{-1} M^{\mathrm{T}})^{-1} M^{\mathrm{T}} [B']^{-1} \frac{\Delta P(\theta_k, V_k)}{V_k}\end{aligned} \tag{8}$$

$$\Delta\theta_k = \Delta\theta'_k + \Delta\theta''_k \tag{9}$$

Similarly, taking the $\Delta V - \Delta Q$, (3) has been developed as

$$\begin{aligned}\Delta V_k = {} & [B']^{-1} \frac{\Delta Q(\theta_{k+1}, V_k)}{V_k} \\ & + [B'']^{-1} M \left([\delta y'']^{-1} + M[B'']^{-1} M^{\mathrm{T}}\right)^{-1} \\ & M^{\mathrm{T}} [B'']^{-1} \frac{\Delta Q(\theta_{k+1}, V_k)}{V_k}\end{aligned} \tag{10}$$

$$\Delta V_k = \Delta V'_k + \Delta V''_k \tag{11}$$

As generator outages are not simulated in the paper, the load balance is not disrupted when a branch outage occurs. Therefore, neither preventive actions nor corrective actions are included as this is not an OPF formulation. As the primary objective and the contribution of the paper are to propose a parallel computing framework for CA and focus on its computational performance, we simply try to see if the power flow can still converge and the system is still secure after a line trips out. Preventive and corrective actions will be included in our future work.

## 3 Master/slave parallel computing

### 3.1 Master/slave MPI program

Using parallel computing to decompose the computational workload to accelerate the execution time, the message passing interface (MPI) paradigm is well suited for handling computations when a task is divided into subtasks. For MPI programs, it is common to have most of the processors for computation, and a few processors or just one processor for task scheduling management. Generally, the manager is called the "master" and the rest are called the "workers" or "slaves" [15].

In this case, firstly, the master usually does all of the I/O work while the slaves complete their I/O work by contacting the master. Hence, the extra I/O overhead is reduced by increasing the numbers of processors. Moreover, it provides the opportunity to monitor the availability of the slaves and achieve load-balancing. Assume that $N$ is the number of processors, $T_s$ is the serial execution time of the program, and $T_p$ is the paralleled execution time of the program by $N$ processors. We here define the speedup $k_{\text{speedup}}$ to evaluate the scalability of parallelism as equation [16]: $k_{\text{speedup}} = T_S/T_N$.

(1) Synchronous and asynchronous message passing: While passing messages synchronously, the program will be stopped until the message is delivered to the destination. While passing messages asynchronously, however, the program will continue forwarding the tasks immediately to carry out processing without blocking other functions. Thus, the I/O and computation can be overlapped, and the efficiency is improved.

(2) Embarrassingly parallelism: Under embarrassingly parallel (massive parallelism) scenarios, the computation is naturally independent among subtasks, but the execution time of each subtask varies. The latest finishing time of a slaves-level subtask determines the response time of the entire application. Thus, the processing time for each slave depends not only on the number of subtasks but on the actual maximum execution time for the assigned tasks.

(3) Static and dynamic parallel computing: The static parallel computing determines scheduling of subtasks by the rank of processors and corresponding task indices. The distribution of tasks is fixed. It is easily implemented, and task assignments are deterministic. However, although the program is massively parallelized, the actual processing time for each task is hardly predictable and is determined

by the input data. Because of such variances in the execution time among slaves, static parallel computing results in a considerable penalty stemming from workload imbalance. On the other hand, dynamic scheduling maintains a work queue, and tasks that are initially assigned by one to each slave and master are instead assigned to a slave by one task at a time while also acknowledging when the previous task is finished. Dynamic task scheduling helps to alleviate such problems but does so at the cost of needing to support synchronized communication, such that the tasks are assigned according to the latest processing states of the slaves instead of according to the rank of processors.

### 3.2 Three-layered master/slave dynamic parallel computing

Whether the program employs static parallel computing with asynchronous message passing or dynamic parallel computing with synchronous message passing, having a single master only creates a bottleneck. Dynamic scheduling should be adopted to minimize variances and thus to gain further scalability, while the synchronization cost should be limited within the capacity of the master.

Hence, we employ a three-layered, hierarchical master/slaves method to approximate such a tradeoff. To address massive parallel computing, based on the above paradigm, tasks are dynamically scheduled to achieve a fine scalability.

The processors are assigned to several groups, and the master in the group communicates with the individual slave processors, receiving results and feedbacks on task assignments. Such a communications workload is within the master's capacity. Therefore, the workload is balanced within groups. Moreover, because the number of slaves in a group is limited such that the number of requests being served by the master is limited as well, both workload scheduling and result collection will less degrade the performance.

In this case, the workload balance and the scalability are guaranteed within groups. As we need to achieve workload balance among the groups to approximate a global workload balance, a coordinator who is dedicated to balancing workloads is used among groups. The tasks are dispatched by a central coordinator to a group master and then passed to the slaves, in this three-layered hierarchical dynamic scheduling structure. The group masters request tasks from the unique central coordinator, in other words, they pull tasks from the central scheduler. On the other hand, the individual slave in a group requests a task from them for one at a time.

Each processor computes its rank to learn its role, the central scheduler and group master or the slave. Also the slave knows its group master. One message contains the information we needed, including the index of the message and the sender processor. According to the tag index we could store the results in the right place. A counter is used to record the current task index, and the bound indicates the range of tasks needed to be processed. Processors update the bounds according to the sending or receiving messages to enable dynamic task assignment. The procedures and the implementation of the three-layered dynamic task scheduling are described as follows:

The central scheduler ( )

1. Check if the counter is out of bound; if it is, then exit.
2. Receive a message indicating which group master sends it.
3. Update the counter by adding the length of package.
4. Send counter to the group master indicating the next task to process.
5. Go back to step 1.

Group master ( )

1. Check whether the counter is out of bound; if it is, then exit.
2. Receive a message indicating it is from which slave.
3. Store the results by index and increase the counter by 1.
4. Send current counter to the slave indicating the next task. Check the counter whether it reaches a threshold for requesting more tasks for further executions; if it does, then send the start index of the package of tasks to the central scheduler; also receive an index from the central scheduler, indicating the new start index for the next package; and such index plus half of length of package as the threshold.
5. Check the counter whether it reaches a threshold to the end of the package of tasks; if it does, then update the start index of the package of tasks; and also update the ending index of the package of tasks.
6. Check whether the results are good within the range of the package of tasks; if it does, then via one-way communication, send sorted results to the central scheduler.
7. Go to step 1 and repeat.

Slaves ( )

1. Check whether the index of tasks is out of bound; if it is, then exit.
2. Process computations.
3. Send results to the group master and receive the index from the group master, indicating the next task for processing.
4. Check whether the start index of the new package is larger than its local one; if it is, than update the value to be the latest one.
5. Go to step 1 and repeat.

In each package, we set the number of tasks to be exactly equal to the number of processors in a group so that the variance of execution times of tasks among groups will not be larger than one of the longest single tasks. Hence, we could achieve approximate workload balance among all processors by sacrificing the task scheduling masters' computation power. Again, since we limit the number of group masters and therefore reduce communication frequency, the synchronization penalty of having a single-point central coordinator is limited. Moreover, once the current package is executed, there will be no waiting interval for slaves in the group, because the group master has performed "pre-fetch" tasks at the threshold of half the number of tasks in the remaining package already. Furthermore, to avoid unnecessary synchronizations, we could perform dynamic scheduling at the end of tasks in the queue, in order to perform static non-blocking task processing and to trigger dynamic scheduling while the static assignment is carried out.

Figure 2 demonstrates this implementation. In the first phase, the tasks are deterministically assigned to slaves and the message passing occurs in a non-blocking way. The second phase starts once the group which is likely to finish the first phase soon reaches a threshold and triggers dynamic scheduling.

## 4 Case study

Two cases are used to demonstrate our method. A case is an IEEE 1,168-bus system to compute 1,473 contingencies and $N-2$ contingency analysis. The other case is an 18,345-branch system, of which parts of the branches are chosen for the $N-2$ contingency analysis. Our experiments are implemented on an Argonne fusion cluster [17], of which each node has a dual quad-core, Intel Xeon quad core Nehalem with 2.53 GHz computation power connected by high-speed infiniband network [18].

While the $PQ$ factor method is used for power flow, the computing cost of each contingency is nearly even. In this case, the impact from workload imbalance to performance is very limited. For example, when preliminary contingency screening is performed for the IEEE 1,168-bus case (with the computation workload of 1,473 contingencies) using 128 processors, the application is finished within 0.4 s, the scalability is nearly linear within a scale-up to 512 processors. Thus, the static parallel computing strategy is preferred for this small system.

Figure 3 demonstrates the performance of all $N-1$ contingencies and a subset of all $N-2$ contingencies (128,000 branch contingencies in total).

In this case, the scale of the problem increases considerably. The limitation of asynchronous message passing emerges. For example, with 1,000 slaves and 1 million information items on outages to be collected directly to a single point, while we adopt non-blocking sending and non-blocking receiving of messages to achieve computing and message passing simultaneously, the cost of maintaining sockets among the processors and handling random values replacing cost over master is considerable. In addition, the busy traffic makes the application unstable and there is considerable scalability loss—it even results in application crashes.

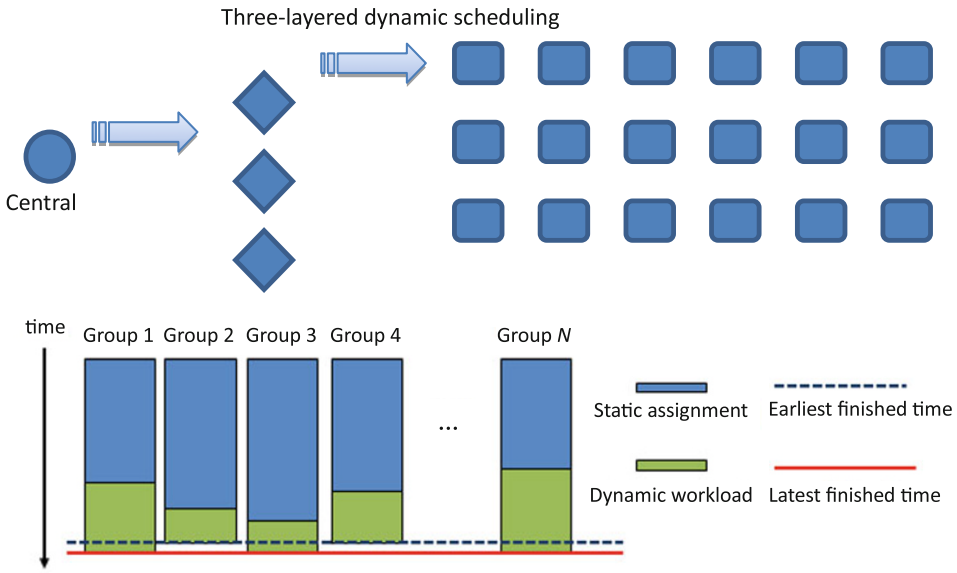


**Fig. 2** Three-layered dynamic scheduling

### 4.1 Static workload computing

We then employ the master/slave two-layered hierarchical paradigm to limit the negative results from having a single master such that the processors are divided into groups and a processor in each group is considered to be the master for collecting the results among the group, which are ordered as a contiguous and integrated array of results. Then the group master delivers the sorted results to the central scheduler. Figure 4 shows that the method performs well while the variance in execution times of performing the subtasks is limited.

The improved method fits well for approximating evenness in the workload among processors, which could be deterministically assigned. However, while performed CA, the computation time for each contingency case varies.

In this case, each elapsed time of contingency calculation is recorded, and variances of elapsed time exist as a result of different iteration times of the CA tasks. The most elapsed times fall between 1.31 and 4.53 s. Figure 5 demonstrates such variances.

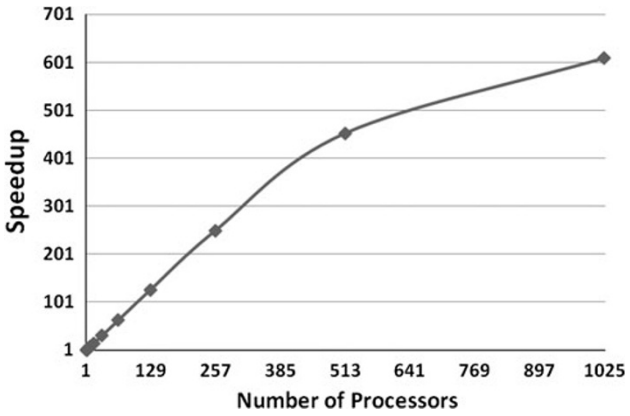


**Fig. 3** Speedup of preliminary CA

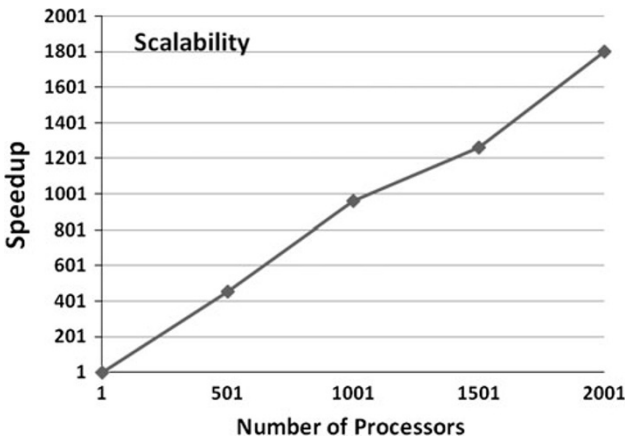


**Fig. 4** Speedup of group-based CA

While we assign tasks statically, the most recently completed time of a task determines the response time of CA. Such an imbalance in CPU elapsed time among processors hurts the speedup of the CA computation, and this drawback will not decrease with the increased scale of the CA problem and also the scale of processors. Hence, the variance goes up along with the scale of the CA problem. The results for a simple experiment adopting static parallel computing and using 4 processors to calculate 50, 100, and 200 contingencies are shown in Table 1. It highlights the impact of workload variance on computational performance.

Hence, the dynamic task scheduling strategy is used to minimize a variance and explore the scalability. In the method, each slave executes its task and sends results to the master. The master uses a counter to determine the index of the tasks to be executed and responds with such a counter to the slave. In this way, we pursue an evenness of execution times among the computational processors.

### 4.2 Two-layered dynamic scheduling

In this case, synchronization overhead is considerable with the increasing number of slaves involved. A single-point coordinator will be a bottleneck to scalability for result receiving and task scheduling. Figure 6 reveals the trade-off between a static non-blocking scheduling strategy and a synchronizing dynamic scheduling strategy. Static parallel computing creates a bottleneck due to having an imbalanced workload, but it minimizes synchronization costs. On the other hand, dynamic scheduling is blocked from achieving further scalability because of the single-point synchronization for task management, although it guarantees workload balance.

We profiled the synchronization costs of 512 slaves scaled with a single task coordinator, as shown in Fig. 7.

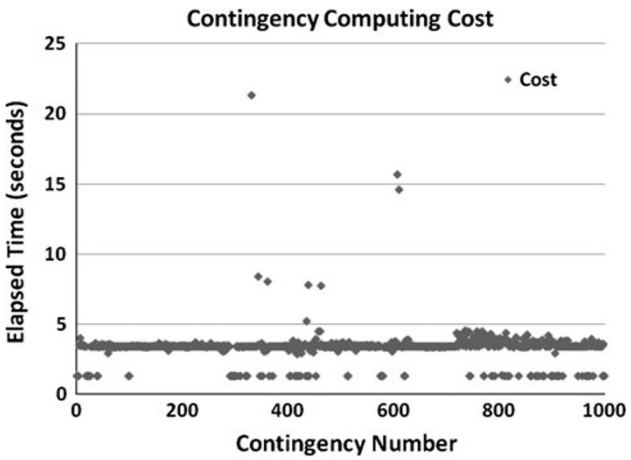


**Fig. 5** Time variances of contingency computing tasks

**Table 1** Impact of workload on computational performance

| Number of contingencies | Finished time (seconds) | Earliest completed time (seconds) |
|---|---|---|
| 50 | 76 | 61 |
| 100 | 164 | 137 |
| 200 | 336 | 300 |

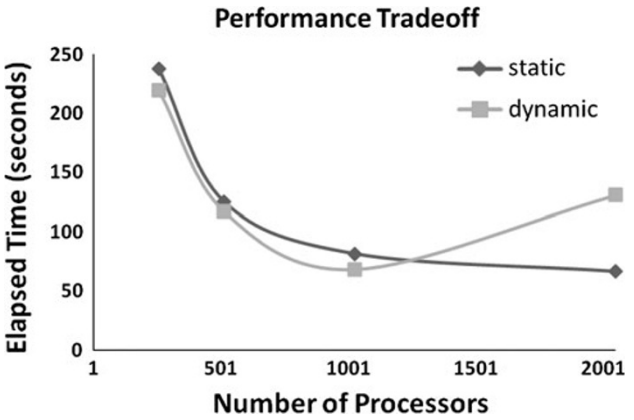


**Fig. 6** Speedup of group-based CA

The spots represent the synchronization costs, which vary among the slaves. Although most of the costs are less than 50 microseconds, some range from 30 microseconds and up to seconds, this is due to the contention for the single-point communication. It blocks the scalability from achieving a further (larger) scale and reveals the trade-off between non-blocking task execution and workload balance.

### 4.3 Three-layered scheduling

In order to demonstrate the potential scalability of the methodology in the $N-X$ CA, 400 contingencies are selected from an 18,368-bus case to process $N-1$ and $N-2$ contingency analysis, or a total of 80,200 contingencies. The performance scale is evaluated when using from 517 processors up to 2,065 processors.

As shown in Fig. 8, we could further scale up to 2,065 processors to solve the problem within 100 s, as opposed to processing for more than 2 days via a single processor. Therefore, we could further scale up to a large CA computation. In this case, although master processors only work for scheduling and somehow reduce scalability, the degree of the reduction of scalability is limited according to the ratio of masters to slaves, as well as the workload balance with limited synchronization costs. The computational cost is approximately distributed in evenness among groups and all processors. Thus, the workload balance is guaranteed to be system-wide workload.

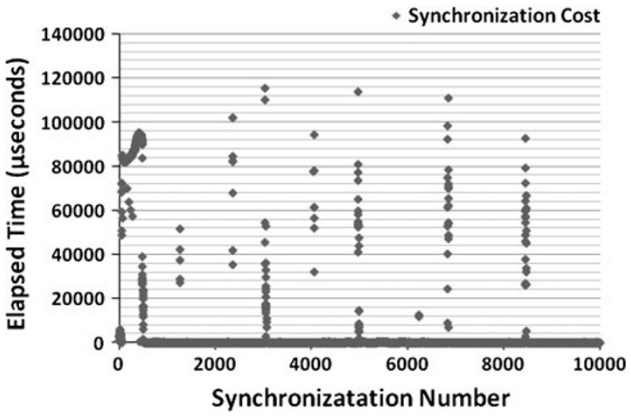


**Fig. 7** Overhead of synchronization

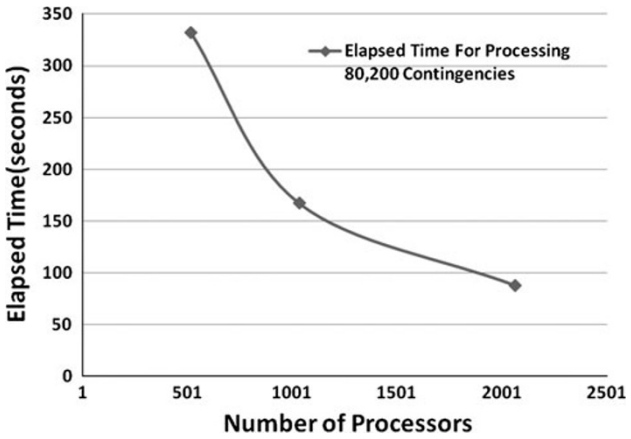


**Fig. 8** Further speedup through three-layered dynamic scheduling

### 4.4 Performance comparison

In our case study, static, two-layered dynamic and the three-layered scheduling for CA are shown. Table 2 demonstrates the results of each case.

Due to different CA cases and CA task execution times, the methods fit for different scenarios. In Scenario 1, the execution time of each task are approximating evenness, the static scheduling method works better, since it avoids synchronization penalties and achieves workload balancing. In Scenario 2, the scale of CA is small, a fine scale of processors work for such computation, since it gains workload balance but pays acceptable synchronization penalties. In Scenario 3, the scale of CA is larger and many more processors are used, the three-layered dynamic scheduling is preferred, because it guarantees workload balancing and limits the synchronization penalties within an acceptable amount.

## 5 Conclusion

A three-layered master/slave hierarchical dynamic scheduling method is presented for massive parallel

**Table 2** Results of each case

| Scenario | Method | Case (bus) | CA task number | Number of processors | Performance (seconds) |
|---|---|---|---|---|---|
| 1 | Static | 1,168 | 128,000 | 2,065 | 15.35 |
| 2 | Two-layered dynamic | 18,345 | 10,000 | 512 | 115.80 |
| 3 | Three-layered dynamic | 18,345 | 80,200 | 2,065 | 87.92 |

computing of CA. We have also demonstrated the potential for processing $N - X$ CA by a large scale method. The variance in the processing time of CA is a concern and is handled by adopting the layered dynamic task scheduling. The results show that the synchronization cost with the proposed method is limited within groups while achieving overall workload balancing.

**Acknowledgments** The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up non-exclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.



## References

[1] Wood A, Wollenberg B (1996) Power generation, operation, and control, 2nd edn. Wiley, New York
[2] Zaborsky J, Whang KW, Prasad K (1980) Fast contingency evaluation using concentric relaxation. IEEE Trans Power Appar Syst 99(1):28–36
[3] Ejebe GC, Wollenberg BF (1979) Automatic contingency selection. IEEE Trans Power Appar Syst 98(1):97–109
[4] Ejebe GC, van Meeteren HP, Wollenberg BF (1988) Fast contingency screening and evaluation for voltage security analysis. IEEE Trans Power Syst 3(4):1582–1590
[5] Fu Y, Shahidehpour M, Li Z (2006) AC contingency dispatch based on security-constrained unit commitment. IEEE Trans Power Syst 21(2):897–908
[6] Monticelli A, Pereira MVF, Granville S (1987) Security-constrained optimal power flow with post-contingency corrective rescheduling. IEEE Trans Power Syst 2(1):175–180
[7] Ezhilarasi GA, Swarup KS (2009) Parallel contingency analysis in a high performance computing environment. In: Proceedings of the 3rd international conference on power systems (ICPS'09), Kharagpur, India, 27–29 Dec 2009, p 6
[8] Gorton I, Huang Z, Chen Y et al (2009) A high-performance hybrid computing approach to massive contingency analysis in the power grid. In: Proceedings of the 5th IEEE international conference on E-science, Oxford, UK, 9–11 Dec 2009, pp 277–283
[9] Huang Z, Chen Y, Nieplocha J (2009) Massive contingency analysis with high performance computing. In: Proceedings of the IEEE PES general meeting (PES'09), Calgary, Canada, 26–30 Jul 2009, p 8
[10] Buyya R (1999) High performance cluster computing: Programming and applications, vol 2. Upper Saddle River, Prentice Hall, pp 19–24
[11] Shao G, Wolski R, Berman F (1998) Performance effects of scheduling strategies for master/slave distributed applications. TR-CS98-598, University of California, San Diego, CA, USA
[12] Sahni S, Vairaktarakis G (1996) The master–slave paradigm in parallel computer and industrial settings. J Glob Optim 9(3/4):357–377
[13] da Silva F, Senger H (2010) Scalability analysis of embarrassingly parallel applications on large clusters. In: Proceedings of the 2010 IEEE International symposium on parallel & distributed processing, Workshops and Ph.D. forum (IPDPSW'10), Atlanta, GA, USA, 19–23 Apr 2010, p 8
[14] Wang X, Song Y, Irving M (2008) Modern power system analysis. Springer, New York
[15] Message Passing Interface (MPI) standard. http://www.mcs.anl.gov/research/projects/mpi/. Accessed 1 March 2012
[16] Wood DA, Hill MD (1995) Cost-effective parallel computing. Computer 28(2):69–72
[17] Laboratory Computing Resource Center. http://www.lcrc.anl.gov/fusion/
[18] InfiniBand. http://www.top500.org/connfam/8

## Author Biographies

**Xi YANG** received his B.S. and M.S. degree in Software Engineering from Jilin University, China. He is currently a Ph.D. student at Computer Science Department of Illinois Institute of Technology, working with Dr. Xian-He Sun. His research interests are evaluation and optimization on distributed computing and parallel computing.

**Cong LIU** got his B.S. and M.S. degrees in Electrical Engineering from Xi'an Jiaotong University, China, in 2003 and 2006, respectively. He received his Ph.D. degree at the Illinois Institute of Technology in Chicago in 2010. Currently, he is working as energy systems computational engineer in the Decision and Information Sciences Division of Argonne National Laboratory. His research interests include numerical computation, optimization and control of power systems.

**Jianhui WANG** received the Ph.D. degree in electrical engineering from Illinois Institute of Technology, Chicago, IL, USA, in 2007. Presently, he is a Computational Engineer with the Decision and Information Sciences Division at Argonne National Laboratory, Argonne, IL, USA. Dr. Wang is the chair of the IEEE Power & Energy Society (PES) power system operation methods subcommittee. He is an editor of the IEEE Transactions on Power Systems, the IEEE Transactions on Smart Grid, an associate editor of Journal of Energy Engineering, an editor of the IEEE PES Letters, and an associated editor of Applied Energy. He is also the editor of Artech House Publishers Power Engineering Book Series and the recipient of the IEEE Chicago Section 2012 Outstanding Young Engineer Award. He is also an affiliate professor at Auburn University.